

Adaptive Mesh Refinement Routines for Overture

Regrid: Adaptive Mesh Refinement Grid Generation

ErrorEstimator: error estimation

InterpolateRefinements: interpolation routines for adaptive grids

Interpolate: interpolate a patch between a fine and coarse grid

David L. Brown
William D. Henshaw

CASC: Centre for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA, 94551
henshaw@llnl.gov
dlb@llnl.gov
<http://www.llnl.gov/casc/Overture>

November 2, 2003

UCRL-MA-140918

Abstract: We describe some of the capabilities in Overture for solving problems with adaptive mesh refinement (AMR). The example program `amrHype`, a simple time-dependent AMR solver, is used to illustrate the AMR approach. The algorithms used to build AMR grids are described. These algorithms are implemented in the class `Regrid`, which makes use of the `Boxlib` library from Lawrence Berkeley Lab. The functions in the class `ErrorEstimator` can be used to compute error estimates of a given solution. This class uses first and second order differences to estimate where the error is large. The `InterpolateRefinement` class is used to interpolate information between different refinement grids, such as interpolating information on all refinement boundaries. The `Interpolate` class implements the actual interpolation formulae for transferring information between coarse and fine grid patches. Convergence results for various test cases are also presented.

Contents

1	Introduction	5
2	A simple time dependent adaptive mesh refinement solver: amrHype	6
3	Adaptive mesh regridding algorithms	8
3.1	Block based aligned grids	8
4	Error Estimation	11
5	Interpolation	13
6	InterpolateRefinements	13
7	Interpolate and InterpolateParameters	16
7.1	Usage	16
8	Accuracy tests	17
9	Regrid Reference Manual	21
9.1	Constructor	21
9.2	getDefaultNumberOfRefinementLevels	21
9.3	getRefinementRatio	21
9.4	outputRefinementInfo	21
9.5	setEfficiency	22
9.6	setGridAdditionOption	22
9.7	setGridAlgorithmOption	22
9.8	setMergeBoxes	22
9.9	setNumberOfBufferZones	22
9.10	setWidthOfProperNesting	22
9.11	setRefinementRatio	23
9.12	setUseSmartBisection	23
9.13	findCut	23
9.14	getBox	23
9.15	buildBox	23
9.16	getBoundedBox	23
9.17	getEfficiency	23
9.18	splitBox	24
9.19	findCut	24
9.20	splitBox	24
9.21	buildTaggedCells	24
9.22	cellCenteredBox	25
9.23	cellCenteredBox	25
9.24	buildProperNestingDomains	25
9.25	printStatistics	26
9.26	buildGrids	26
9.27	regrid	26
9.28	regrid	27

9.29	regridAligned	27
9.30	splitBoxRotated	28
9.31	merge	28
9.32	regridRotated	28
9.33	displayParameters	29
9.34	get	29
9.35	put	29
9.36	update	29
10	ErrorEstimator Reference Manual	30
10.1	Constructor	30
10.2	setDefaultNumberOfSmooths	30
10.3	setScaleFactor	30
10.4	setTopHatParameters	30
10.5	setWeights	30
10.6	computeErrorFunction	30
10.7	computeFunction	31
10.8	31
10.9	interpolateAndApplyBoundaryConditions	31
10.10	computeErrorFunction	31
10.11	computeErrorFunction	32
10.12	computeErrorFunction	32
10.13	smoothErrorFunction	32
10.14	plotErrorPoints	33
10.15	get	33
10.16	put	33
10.17	update	33
11	InterpolateRefinements Reference Manual	34
11.1	Constructor	34
11.2	setOrderOfInterpolation	34
11.3	setNumberOfGhostLines	34
11.4	intersects	34
11.5	getIndex	34
11.6	getIndex	35
11.7	35
11.8	buildBox	35
11.9	buildBaseBox	35
11.10	interpolateRefinementBoundaries	36
11.11	interpolateCoarseFromFine	36
11.12	get	36
11.13	put	36
11.14	interpolateRefinementBoundaries	36
11.15	interpolateCoarseFromFine	37

12 Interpolate Reference Manual	38
12.1 Interpolate default constructor	38
12.2 Interpolate constructor	38
12.3 initialize	38
12.4 interpolateFineToCoarse	39
12.5 interpolateCoarseToFine	39
13 InterpolateParameters Reference Manual	40
13.1 InterpolateParameters default constructor	40
13.2 InterpolateParameters destructor	40
13.3 setAmrRefinementRatio	40
13.4 setInterpolateType	40
13.5 numberOfDimensions	41
13.6 setInterpolateOrder	41
13.7 setGridCentering	41
13.8 setUseGeneralInterpolationFormula	41
13.9 interactivelySetParameters	42
13.10 amrRefinementRatio	42
13.11 gridCentering	42
13.12 useGeneralInterpolationFormula	42
13.13 interpolateType	42
13.14 interpolateOrder	42
13.15 numberOfDimensions	42

1 Introduction

The adaptive mesh refinement (AMR) approach adds new refinement grids where the error is estimated to be large. The refinement grids are usually aligned with the underlying base grid (the refinement is done in index-space). The refinement grids are arranged in a hierarchy, with the base grids belonging to level zero, the next grids being added to level 1 and so on. Grids on level m are refined by a **refinement ratio** r , (usually 2 or 4) from the grids on level $m - 1$. The grids are normally **properly nested** so that a grid on level m is completely contained in the grids on the coarser level $m - 1$.

Regridding : Every few steps (usually every r (refinement ratio) steps) the adaptive grid will be rebuilt. The `Regrid` class can be used to rebuild the AMR grid. On an overlapping grid, the `updateRefinement` function in the `Ogen` overlapping grid generator must also be called. This latter function will compute the overlapping grid interpolation points for the refinement grids. When a new grid has been generated, the solution must be transferred from the old grid to the new grid. Solution values on the new grid prefer to interpolate from the finest level grid available on the old grid.

Interpolation : During every time step, the ghost boundaries of a refinement grid on level m are interpolated from refinement grids on level m , or level $m - 1$ if there are no level m grids available. Coarse grid points that are covered by fine grids are interpolated from the finer grid. On an overlapping grid, refinement grids that hit the overlapping grid interpolation boundary will interpolate from grids belonging to another base grid.

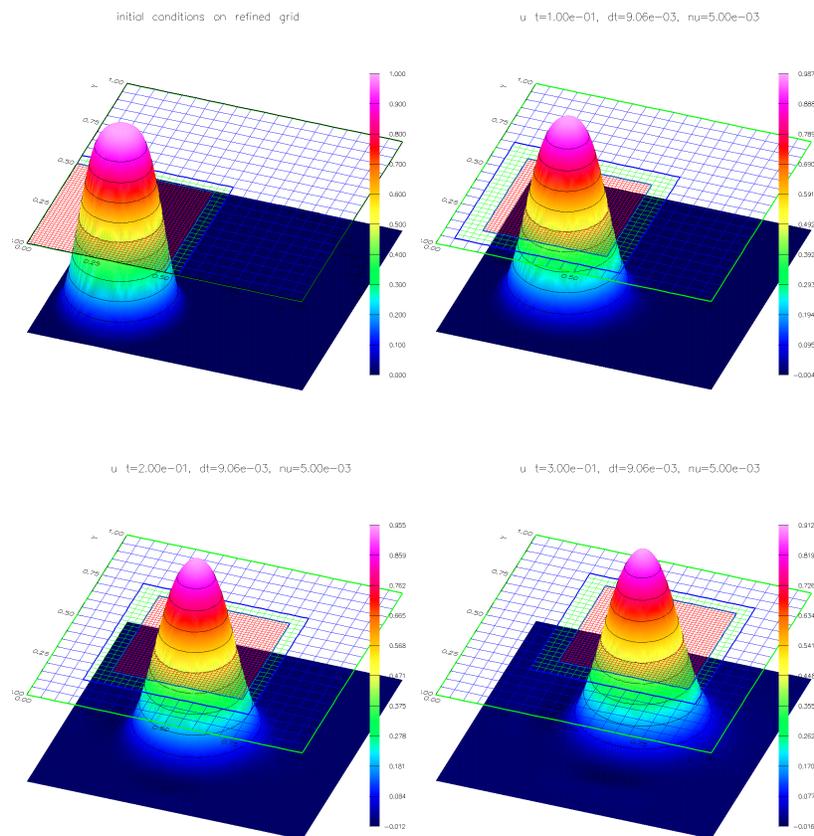


Figure 1: Results from `hypeAmr`, solving a convection diffusion equation with adaptive mesh refinement.

2 A simple time dependent adaptive mesh refinement solver: amrHype

The program `amrHype.C` found in `Overture/primer/amrHype.C` solves a convection diffusion equation using adaptive mesh refinement (AMR). The equation

$$\begin{aligned}u_t + au_x + bu_y &= \nu \Delta u \\ u(\mathbf{x}, 0) &= u_0(\mathbf{x})\end{aligned}$$

is advanced with a fourth-order accurate Runge-Kutta time stepping algorithm. **NOTE:** for simplicity, a single time step is used on all grids.

Here is a pseudo-code outline of the AMR algorithm implemented in `amrHype`.

```
amrHype( $g, l_f$ )
// amrHype: time step a convection-diffusion equation
 $l_f$  : finest level to build
 $g, g_n$  : old and new grids
 $u$  : grid function holding the solution
 $u_0$  : initial condition function
 $r_f$  : refinement factor, e.g. 2 or 4
{
   $g$  : Initial grid
   $u(g) := u_0(g)$ ;      assign initial conditions
  // build the initial AMR grid, add one new level at a time.
  for  $l = 1, \dots, l_f$ 
     $e := \text{estimateError}(u)$ 
     $g_n = \text{regrid}(g, e, r_f, l)$ ;      build a new grid with  $l$  levels
     $u(g_n) := u_0(g_n)$ ;      re-assign initial conditions
     $g := g_n$ ;
  end

   $i := 0$ ;  $t = 0$ ;
   $\Delta t := \text{computeTimeStep}(g)$ ;
  while  $t < t_f$ 
    if ( $i \bmod r_f == 0$ )
      // regrid every  $r_f$  steps
       $e := \text{estimateError}(u)$ 
       $g_n = \text{regrid}(g, r_f, e, l)$ ;
      interpolateToNewGrid( $u$ );
       $g := g_n$ ;
       $\Delta t := \text{computeTimeStep}(g)$ ;
    end

    timeStep( $u$ );
    interpolate( $u$ );
    applyBoundaryConditions( $u, t$ );
     $t := t + \Delta t$ ;
  end
}
```

Every few steps a new AMR grid is computed, based on an estimate of the error. The solution must then be interpolated from the old AMR grid to the new AMR grid. The AMR algorithm is implemented with the help of the following classes

ErrorEstimator : class used to compute error estimates.

Regrid : class used to build a new AMR grid.

InterpolateRefinements : used to

- interpolate from one AMR grid to a second AMR grid,
- interpolate ghost-boundaries of refinement grids
- interpolate coarse grid points that are hidden by refinement grids.

This class in turn uses the **Interpolate** class which knows how to interpolate refinement patch points from a coarser grid.

3 Adaptive mesh regridding algorithms

The basic block-structured adaptive mesh refinement regridding algorithm can be found in the thesis of Berger[3].

The basic idea for building new refinement grids is illustrated in figure (2). The goal is to cover a set of tagged cells by a set of non-overlapping boxes. For efficiency, the boxes are not allowed to become too small, nor must they be too empty. Each box satisfies an ‘efficiency’ condition where the ratio of tagged cells to untagged cells must be larger than some `efficiencyFactor` $\approx .7$.

1. Given an estimate of the error, tag cells where the error is too large. Fit a box to enclose the tagged cells.
2. Recursively sub-divide the box. Split the box in the longest direction, at a position based on the histogram formed from the sum of the number of tagged cells per row or column. The exact formula is given later.
3. After splitting the box, fit new bounding boxes to each half and repeat the process. Continue until the efficiency of the box is larger than the `efficiencyFactor`.

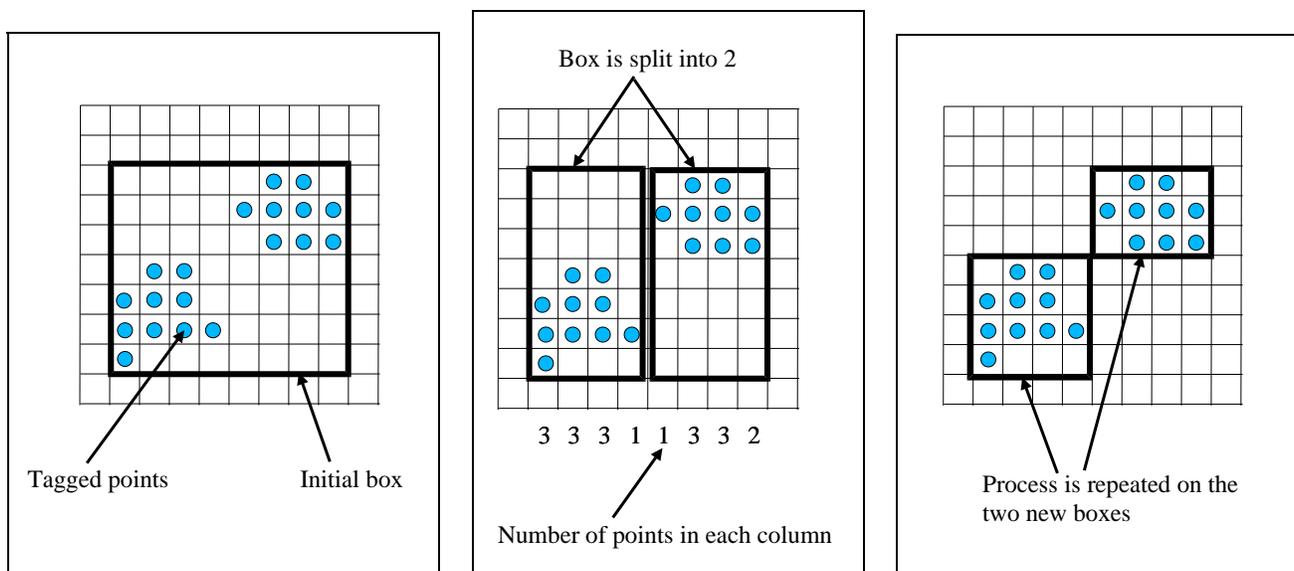


Figure 2: The 3 basic steps in regridding are (1) tag error cells and enclose in a box, (2) split the box into 2 based on a histogram of the column or row sums of tagged cells, (3) fit new boxes to each split box and repeat if the ratio of tagged to untagged cells is too small.

3.1 Block based aligned grids

This algorithm is based on the regridding procedure from LBL, found in their HAMR package. This algorithm is based on the algorithm from Bell-Berger-Collela-Rigoustos-Saltzman-Welcome [5, 4, 2, 1].

The basic idea is to recursively bisect the region of tagged points until the resulting patches satisfy an efficiency criteria.

Here is the algorithm

Algorithm 3.1 A block AMR grid generator:

regrid(l_b, l_f, G_l, e)

l_b : base level, this level and below do not change

l_f : fine level, build this new level

e : errors defined on all grids for levels $l = l_b, \dots, l_f - 1$

G_l : Set of boxes for level l

\overline{G}_l : complement of G_l with respect to some large bounding box.

E_m : Operator that expands each box by m cells;

R_r : Operator that refines each box by ratio r

n_b : Number of buffer zones

{

Build proper nesting domains

$l := l_b$

$P_l := \overline{E_m G_l}$: Boxes defining the properly nested region

for $l = l_b + 1, \dots, l_f - 1$

$P_l := \overline{E_m R_r P_{l-1}}$

end

Build levels from finest level down

for $l = l_f, l_f - 1, \dots, l_b + 1$

Build a list of tagged cells for all grids at this level

tag : List that will hold indices of tagged cells

for each grid at this level

$tag := tag \cup (error > tol)$: add points where error is large

$tag :=$ add n_b neighbours of tagged cells

if $l < l_f$

add cells that lie beneath tagged cells on level $l+1$

end

end

$b_0 := \mathbf{buildBox}(tag)$: build a box around all tagged cells

Recursively sub-divide the box

$B := \emptyset$: The set that will contain the refinement boxes

split(b_0, B)

$B := \mathbf{merge}(B)$

end

}

Recursively sub-divide a box:

```

split(box  $b$ , boxSet  $B$ )
{
  if  $b$  is efficient or too small
    if  $b \subset P_l$ 
       $b$  lies in the region of proper nesting,  $P_l$ 
       $B := B \cup b$ ;
    else
       $B_0 := b \cap P_l$  : set of boxes for intersection
       $B := B \cup B_0$  : add boxes from  $B_0$ 
    end
  else
    divide  $b$  into 2 boxes, split along the longest side
     $\{b_1, b_2\} := \text{subDivide}(b)$ 
    split( $b_1, B$ )
    split( $b_2, B$ )
  end
}

```

Sub-divide a box in a smart way:

```

subDivide(box  $b, \alpha$ )
 $\alpha$  : divide box along this axis
{
   $h_i :=$  number of tagged points with  $i_\alpha = i, i = 0, \dots, N$ 
   $m := N/2$ 
   $Z := \{i | h_i \equiv 0\}$ 
  if  $Z \neq \emptyset$ 
    split at the middle most point where the  $h_i = 0$ 
     $j := m + \min_{i \in Z} |i - m|$ 
  else
    split at the middle most point where the 2nd derivative changes sign,
    and the 3rd derivative is largest in magnitude
     $d_i := h_{i+1} - 2h_i + h_{i-1}$ 
     $S := \{i | d_{i-1}d_i < 0 \text{ and } |d_i - d_{i-1}| \text{ is maximal}\}$ 
    if  $S \neq \emptyset$ 
       $j := m + \min_{i \in S} |i - m|$ 
    else
       $j := m$ 
    end
  end
  split box  $b$  at index  $j$  creating boxes  $b_1$  and  $b_2$ 
  return{ $b_1, b_2$ }
}

```

Figure (3) illustrates the proper nesting region (P_l in the above algorithm) for a set of boxes. The proper nesting region for a set of boxes is the region interior to the set of boxes which lies a certain buffering distance from the coarse grid boundary. The proper nesting region defines the region into which new refinement patches are allowed to lie.

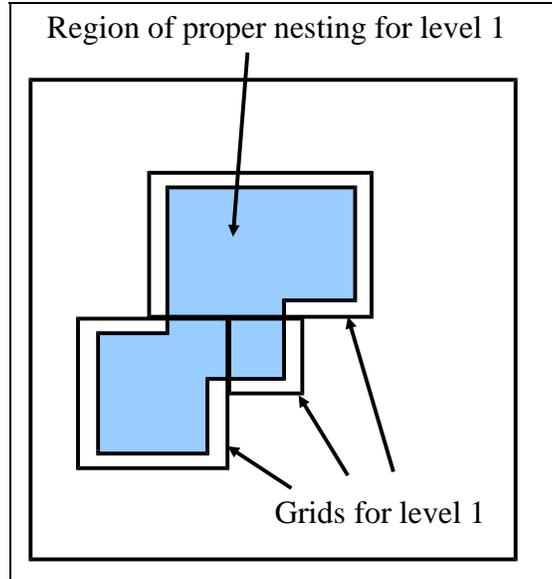


Figure 3: The proper nesting region for a set of boxes is the region interior to the set of boxes which lies a certain buffering distance from the coarse grid boundary. The proper nesting region defines the region into which new refinement patches are allowed to lie.

4 Error Estimation

The `ErrorEstimator` class can be used to estimate errors in a solution.

One way to estimate errors is to use a weighted combination of first and second differences. The error in component n of a vector \mathbf{u} is estimated as

$$e_n = \frac{1}{d} \sum_{m=1}^d \frac{c_1}{s_n} \|\Delta_0^m \mathbf{u}_n\| + \frac{c_2}{s_n} \|\Delta_+^m \Delta_-^m \mathbf{u}_n\|$$

where s_n is a scale factor.

The error function is normally smoothed a few times using an under-relaxed Jacobi iteration. After each smoothing step the error is interpolated to neighbouring grids.

Smoothing the error serves the purpose of propagating errors to nearby cells. On an overlapping grid this will cause refinement grids to be added properly as a feature crosses from one base grid to another base grid. By the time a sharp feature reaches an overlapping grid interpolation boundary, refinement grids should already have been created on the nearby base grids.

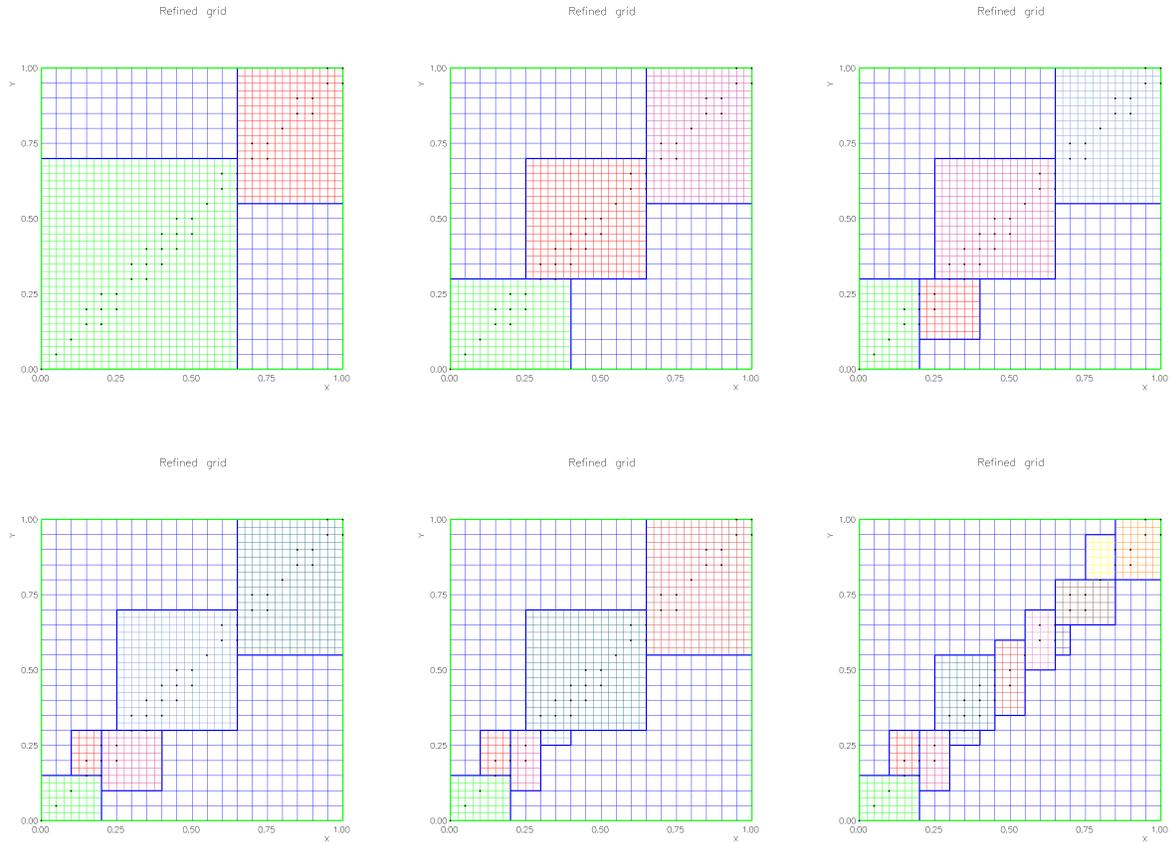


Figure 4: Steps in the AMR regridding algorithm. The grid is shown after 1,2,3,4,5 and all steps. The tagged error points are marked as black dots.

```

estimateError( $g, \mathbf{u}, n_s, e$ )
// amrHype: time step a convection-diffusion equation
 $g$  : adaptive grids
 $\mathbf{u}$  : grid function holding the solution, with components  $\mathbf{u}_n$ 
 $n_s$  : number of times to smooth
 $e$  : estimated error (output)
{
  // get error estimate:
   $e(g) := \frac{1}{d} \sum_{m=1}^d \frac{c_1}{s_n} \Delta_0^m \mathbf{u}_n + \frac{c_2}{s_n} \Delta_+^m \Delta_-^m \mathbf{u}_n;$ 

  smooth the error estimate:
  interpolate( $e$ );
  for  $i = 1, \dots, n_s$ 
    smooth( $e$ );
    interpolate( $e$ );
  end
}

```

5 Interpolation

The `interpolate` function in the `Interpolant` class can be used to interpolate both overlapping grid interpolation points and AMR interpolation points. When the grid is created with **implicit** interpolation the interpolation equations are coupled. The coupled equations are either solved with a sparse matrix solver or they can be solved by iteration (this is an option that can be set with the `Interpolant`). In general the solution of the implicit equations is expensive and it is preferable to use explicit interpolation.

With explicit interpolation the interpolation points on the base grids are uncoupled. However, with AMR grids there may be some implicit coupling between overlapping grid interpolation points and AMR interpolation points (see the example in figure ??). We can avoid iterating to solve the interpolation equations provided we solve the equations in the correct order.

Here is the algorithm we use for explicit interpolation. On input we assume that the solution is known at all interior and boundary points except for interpolation points.

This is not quite correct yet*.

Interpolant::explicitInterpolate(*u*)

// interpolate overlapping grid and AMR interpolation points

u : grid function holding the solution

```
{
  u.periodicUpdate
  interpRefinements.interpolateCoarseFromFine(u) : // interpolate coarse grid points hidden by refinement
  u.applyBoundaryCondition(extrapolateRefinementBoundaries)
  interpolateOverlappingGridPoints
  u.applyBoundaryCondition(extrapolate, allBoundaries)
  u.applyBoundaryCondition(extrapolateInterpolationNeighbours)
  u.finishBoundaryConditions
  interpRefinements.interpolateRefinementBoundaries(u);
}
```

6 InterpolateRefinements

The `InterpolateRefinements` class is used to

- interpolate a grid function from one AMR grid to a second AMR grid,
- interpolate ghost-boundaries of refinement grids
- interpolate coarse grid points that are hidden by refinement grids.

Here is the algorithm for interpolating a grid function on one AMR grid from a grid function belonging to a second AMR grid. This function is called after a new AMR grid has been created in order to transfer the solution from the old grid to the new. The basic idea is to interpolate values for a new grid function on level l from old grid function values on levels l or below.

```

interpolateRefinements(  $g^{\text{old}}, u^{\text{old}}, g, u$  )
// interpolate u on grid g from  $u^{\text{old}}$  on grid  $g^{\text{old}}$ 
 $g^{\text{old}}, u^{\text{old}}$  : old grid and grid function
 $g, u$  : new grid and grid function
{
  for each base grid  $b_g$ 
     $u[b_g] = u^{\text{old}}[b_g]$ ;
  end
  for each refinement level  $l = 1, \dots, l_f$ 
     $g_l := g.\text{refinementLevel}[l]$ ;
    for each component grid in  $g_l$ 
      // Interpolate  $u_k$  from  $u^{\text{old}}$ 
      for  $m = l, l - 1, \dots, 1$ 
        Try to interpolate from a grid on level m
         $g_m^{\text{old}} := g^{\text{old}}.\text{refinementLevel}[m]$ ;
        for each component grid in  $g_m^{\text{old}}$ 
          Fill in any points of intersection
          Fill a mask to keep track of which points have been filled
          if all points have been interpolated
            break; we are done with this grid
          end
        end
      end
    end
  end
}

```

Here is the algorithm for interpolating the refinement boundaries. We use the interpolation functions available in the `Interpolate` class to actually do the interpolation.

interpolateRefinementBoundaries(g, u)

// interpolate ghost boundaries on u

g, u : grid and grid function

{

for each refinement level $l = l_s, \dots, l_f$

$g_l := g.\text{refinementLevel}[l]$;

for each component grid in g_l

 // Interpolate ghost boundaries of u_k

 // First interpolate all ghost boundaries from grids at the coarser level

for each component grid $g_{l-1,k}$, in g_{l-1}

 Intersect ghost boundary with grid g_k .

 interpolate.**interpolate**(u, I);

end

 // Now interpolate ghost boundaries from other the grids at the same level

for each component grid $g_{k,l}$, in g_l

 Intersect ghost boundary with grid g_k .

 copy points of intersection

end

end

end

}

7 Interpolate and InterpolateParameters

In this section we describe the `Interpolate` class and its companion class, the `InterpolateParameters` class, that provide interpolation routines for adaptive mesh refinement and multigrid algorithms. Specifically, interpolation routines are provided that transfer data between fine and coarse meshes in an adaptive mesh refinement or multigrid hierarchy. These classes do not provide interpolation for the case where the source and target functions live on grids whose underlying `Mapping`'s are different.

The `Interpolate` class currently supports polynomial interpolation of arbitrary order on `vertexCentered` grids. The `InterpolateParameters::interpolateOrder` parameter determines the interpolation order. It also supports arbitrary refinement factors in each direction. The refinement factors can be different in different directions. The `InterpolateParameters::amrRefinementFactor` array is used to set the refinement factor.

7.1 Usage

Usually, one sets up the desired interpolation parameters using the `InterpolateParameters` class `set` functions. A convenient interactive way to set these parameters is demonstrated in the code below, using the `InterpolateParameters::interactivelySetParameters()` and `InterpolateParameters::display()` functions. The `InterpolateParameters` object is then passed to the `Interpolate` class through the `Interpolate::initialize()` function. The functions `Interpolate::interpolateFineToCoarse` and `Interpolate::interpolateCoarseToFine` are used to perform the interpolation. The region on the target grid that is to receive interpolated values is described using an array of `A++ Indexobjects`, stored in `Iv`.

The sample code `Overture/tests/testInterpolate.C` demonstrates the two-dimensional interpolation from coarse to fine grids.

The sample code `Overture/tests/testInterpolateFineToCoarse.C` demonstrates two-dimensional interpolation from fine to coarse grids

8 Accuracy tests

The test code `hype.C` solves a convection-diffusion equation on adaptive grids with Runge-Kutta time stepping. It is similar in structure to `amrHype` but it has extra code for measuring errors and performance.

Here are some accuracy results from running `hype`.

	40×40	$20 \times 20 + (r = 2)^1$
error	$8.75e - 02$	$8.75e - 02$
grid points	2025	1738
time (s)	6.0	10.6

Effective resolution 40×40

	80×80	$20 \times 20 + (r = 2)^2$	$20 \times 20 + (r = 4)^1$
error	$2.20e - 02$	$2.20e - 02$	$2.20e - 02$
grid points	7225	3882	3827
time (s)	48.0	72.5	57.7

Effective resolution 80×80

Table 1: Computed errors at $t = .5$ for a pulse crossing square.

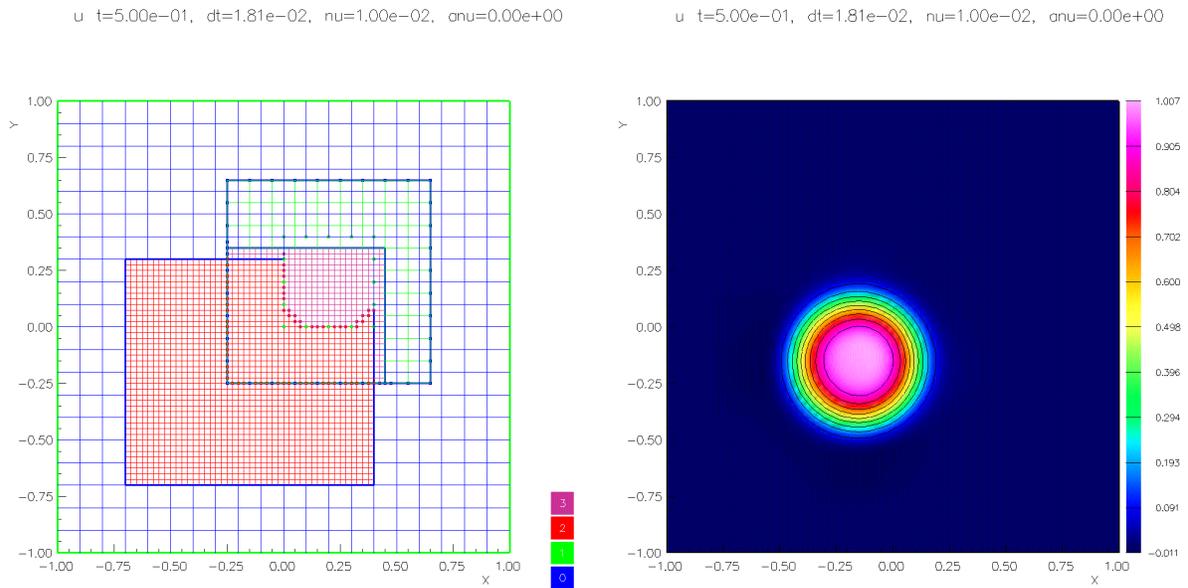


Figure 5: Results from hype, square in a square, $r = 4$, 2 levels, $t = .5$.

	40×40	$20 \times 20 + (r = 2)^1$
error	$1.36e - 01$	$1.36e - 01$
grid points	2386	2542
time (s)	4.4	9.4

Effective resolution 40×40

	80×80	$20 \times 20 + (r = 2)^2$	$20 \times 20 + (r = 4)^1$
error	$3.61e - 02$	$3.61e - 02$	$3.61e - 02$
grid points	8314	5970	5886
time (s)	30.7	59.5	43.6

Effective resolution 80×80

	160×160	$20 \times 20 + (r = 2)^3$	$40 \times 40 + (r = 2)^2$	$40 \times 40 + (r = 4)^1$
error	$8.91e - 03$	$8.91e - 03$	$8.91e - 03$	$8.91e - 03$
grid points	30946	15444	14156	13574
time (s)	394.8	412.6	367.0	283.2

Effective resolution 160×160 Table 2: Computed errors at $t = 1$. for a pulse crossing a square inside a square.

	$20 \times 20 + (r = 4)^1$			
error	$3.61e - 02$	$3.61e - 02$	$3.61e - 02$	$3.62e - 02$
grid points	5886	5369	4984	4434
time (s)	43.4	42.2	42.8	41.9
ϵ	$1.00e - 02$	$5.00e - 02$	$1.00e - 02$	$5.00e - 02$
buffer	2	2	1	1

Effective resolution 80×80 Table 3: Computed errors at $t = 1$. for a pulse crossing a square inside a square.

	$40 \times 40 + (r = 4)^1$	$40 \times 40 + 4^1$	$40 \times 40 + 4^1$	$40 \times 40 + 4^1$
error	$8.90e - 03$	$1.53e - 02$	$1.46e - 02$	$8.81e - 03$
grid points	11777	10325	12079	11026
time (s)	268.2	244.3	282.8	307.7
ϵ	$1.00e - 02$	$5.00e - 02$	$1.00e - 02$	$5.00e - 03$
buffer	2	2	1	1

Effective resolution 160×160 Table 4: Computed errors at $t = 1$. for a pulse crossing a square inside a square.

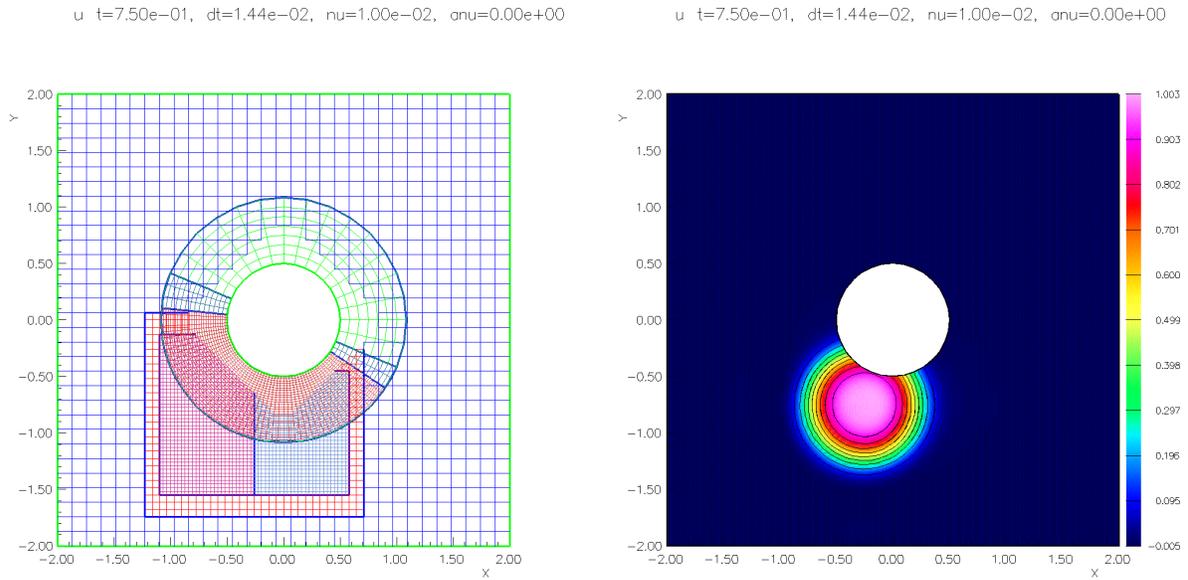


Figure 6: Results from hype, circle in a square, $r = 2, 3$ levels, $t = .75$.

	62×62	$31 \times 31 + (r = 2)^1$
error	$7.13e - 02$	$7.33e - 02$
grid points	5662	3542
time (s)	9.5	14.5

Effective resolution 62×62

	124×124	$31 \times 31 + (r = 2)^2$	$31 \times 31 + (r = 4)^1$
error	$1.98e - 02$	$2.07e - 02$	$2.07e - 02$
grid points	20498	7820	6560
time (s)	91.3	92.2	64.4

Effective resolution 124×124

Table 5: Computed errors at $t = 1$. for a pulse crossing a circle inside a square.

9 Regrid Reference Manual

9.1 Constructor

Regrid()

Description: Use this class to build adaptive mesh refinement grids.

9.2 getDefaultNumberOfRefinementLevels

int

getDefaultNumberOfRefinementLevels() const

Description: Return the default number of refinement levels.

9.3 getRefinementRatio

int

getRefinementRatio() const

Description: Return the refinement ratio.

9.4 outputRefinementInfo

int

**outputRefinementInfo(GridCollection & gc,
 const aString & gridFileName,
 const aString & fileName)**

Description: This function will output a command file for the "refine" test code.

gc(input) : name of the grid.

refinementRatio (input) : refinement ratio.

gridFileName (input) : grid file name, such as "cic.hdf". This is not essential, but then you will have to edit the comamnd file to add the correct name.

fileName (input) : name of the output command file, such as "bug.cmd" The output will be a file of the form

```
choose a grid
  cic.hdf
add a refinement
  0 1 4 10 12 15
add a refinement
  0 1 3 10 15 19
add a refinement
  1 1 12 16 0 7
add a refinement
  1 1 16 20 3 7
```

9.5 setEfficiency

void
setEfficiency(real efficiency_)

Description: Set the regridding efficiency, the ratio of tagged to un-tagged points.

efficiency_ (input) : regridding efficiency.

9.6 setGridAdditionOption

void
setGridAdditionOption(GridAdditionOption gridAdditionOption_)

Description: New grids can be added as refinement grids or as additional base grids.

9.7 setGridAlgorithmOption

void
setGridAlgorithmOption(GridAlgorithmOption gridAlgorithmOption_)

Description: Specify the algorithm to use.

gridAlgorithmOption_ (input) : one of aligned or rotated

9.8 setMergeBoxes

void
setMergeBoxes(bool trueOrFalse =true)

Description: Indicate whether boxes should be merged.

9.9 setNumberOfBufferZones

void
setNumberOfBufferZones(int numberOfBufferZones_)

Description: Specify the number of buffer zones to increase the tagged area by. The boundary of refinement grids will be this number of **coarse grid cells** away from the boundary of the next coarser level. Note that numberOfBufferZones_i=1 since we always transfer node centred errors to surrounding cells.

numberOfBufferZones_ (input) :

9.10 setWidthOfProperNesting

void
setWidthOfProperNesting(int widthOfProperNesting_)

Description: Specify the number of buffer zones between grids on a refinement level and grids on the next coarser level. The value for widthOfProperNesting should be greater than or equal to zero.

9.11 setRefinementRatio

void
setRefinementRatio(int refinementRatio_)

Description: Set the refinement ratio.

9.12 setUseSmartBisection

void
setUseSmartBisection(bool trueOrFalse =true)

Description: Indicate whether the smart bisection routine should be used.

9.13 findCut

int
findCut(int *hist, int lo, int hi, CutStatus &status)

Description: Code taken from HAMR from LBL.

9.14 getBox

BOX
getBox(const intArray & ia)

Description: Build the smallest box that covers a list of tagged cells.

9.15 buildBox

BOX
buildBox(Index Iv[3])

Description: Build a box from 3 Index objects.

9.16 getBoundedBox

BOX
getBoundedBox(const intArray & ia) , MappedGrid & mg)

Description: Build the smallest box that covers a list of tagged cells BUT that is also at least minimum-BoxWidth points wide in each direction.

9.17 getEfficiency

real
getEfficiency(const intArray & ia, const BOX & box)

Description: return the efficiency of a box, the ratio of tagged cells to non-tagged cells.

9.18 splitBox

int

fixPeriodicBox(MappedGrid & mg, BOX & box, const intArray & ia, int level)

Description: Fix a box that is used for a periodic grid such as an Annulus.

Find an appropriate place to split the box along the periodic direction and then shift the tagged points in the array ia that one side of the cutPoint so the other side of the branch cut. When the box is split, this will allow refinement patches to cross the branch cut.

9.19 findCut

int

findCutPoint(BOX & box, const intArray & ia, int & cutDirection, int & cutPoint)

Description: Find the best place to split the box

box (input) : box to possibly split

ia (input) : array of tagged cells.

cutDirection (input/output): on input: if > 0 , cut the box in this direction, otherwise choose a direction to cut the box. On output: box was cut in this direction

cutPoint (output): box was cut at this point.

9.20 splitBox

int

splitBox(BOX & box, const intArray & ia, BoxList & boxList, int refinementLevel)

Description: Split a box into two if it does not satisfy the efficiency criterion. This function then calls itself recursively.

box (input) : box to possibly split

ia (input) : array of tagged cells.

boxList (input/output) :

refinementLevel (input) :

9.21 buildTaggedCells

int

buildTaggedCells(MappedGrid & mg,
 intMappedGridFunction & tag,
 const realArray & error,
 real errorThreshold,
 bool useErrorFunction,
 bool cellCentred = true)

Description: Build the integer flag array (tags) of points that need to be refined on a single grid. Increase the region covered by tagged by the specified buffer zone.

mg (input) : build tags for this grid.

tag (output) : tagged cells go on this grid.

error (input): user defined error function (only used if useErrorFunction==true).

errorThreshold (input) : tag cells where the error is larger than this value.

useErrorFunction (input) : if false, the tag array is already set and we ignore the error array.

9.22 cellCenteredBox

Box

cellCenteredBox(MappedGrid & mg, int ratio =1)

Description: Build a cell centered box from a MappedGrid.

9.23 cellCenteredBox

Box

cellCenteredBaseBox(MappedGrid & mg)

Description: Build a cell centered box from a MappedGrid on level=0.

We expand the box on the base level to include ghost points on interpolation boundaries, since we need to allow refinement patches to extend into the interpolation region.

On a periodic grid we extend the box in the periodic direction since we should never need to restrict refinements in this direction

9.24 buildProperNestingDomains

int

**buildProperNestingDomains(GridCollection & gc,
 int baseGrid,
 int refinementLevel,
 int baseLevel,
 int numberOfRefinementLevels)**

Description: Build a list of boxes that covers the portion of the domain where we are allowed to add refinement grids, in order to ensure proper nesting of grids.

properNestingDomain[level] : a list of boxes defining the allowable region where refinement grids at level+1 can be added. The allowable region will be widthOfProperNesting inside the refinement grids at level.

complementOfProperNestingDomain[level] : the set complement of properNestingDomain[level].

9.25 printStatistics

```
int
printStatistics( GridCollection & gc, FILE *file = NULL,
                int *numberOfGridPoints =NULL)
```

Description: Print statistics about the adaptive grid such as the number of grids and grid points at each refinement level.

file (input): write to this file (if specified)

numberOfGridPoints (input) : return the total number of grid points (if specified)

9.26 buildGrids

```
int
buildGrids( GridCollection & gcOld,
            GridCollection & gc,
            int baseGrid,
            int baseLevel,
            int refinementLevel,
            BoxList *refinementBoxList,
            IntegerArray **gridInfo )
```

Description: Add grids to a GridCollection that can be found in the list of boxes for each refinement level.

gcOld (input) : old grid

gc (output) : new grid

baseGrid (input) : the base grid is gcOld[baseGrid]

baseLevel (input) : this level and below have not been changed.

refinementBoxList (input) : lists of boxes to be added as grids.

gridInfo (input) : holds list of boxes for the new optimized method

9.27 regrid

```
int
regrid( GridCollection & gc,
        GridCollection & gcNew,
        realGridCollectionFunction & error,
        real errorThreshold,
        int refinementLevel = 1,
        int baseLevel = -1)
```

Description: Build refinement grids to cover the "tagged points" where the error is greater than an errorThreshold.

Use a bisection approach to build the grids.

gc (input) : old grid

gcOld (output) : new grid (must be a different object from gc).

error (input): user defined error function

errorThreshold (input) : tag cells where the error is larger than this value.

refinementLevel (input) : highest refinement level to build, build refinement levels from baseLevel+1,...,refinementLevel

baseLevel (input) : this level and below stays fixed, by default baseLevel=refinementLevel-1 so that only one level is rebuilt.

9.28 regrid

int

```
regrid( GridCollection & gc,
        GridCollection & gcNew,
        intGridCollectionFunction & errorMask,
        int refinementLevel = 1,
        int baseLevel = -1)
```

Description: Regrid based on an error mask.

gc (input) : grid to regrid.

gcNew (input) : put new grid here (must be different from gc)

errorMask (input/output) : $\neq 0$ at points to refine, 0 otherwise. Note: this function may be changed on output. It will reflect the actual points refined, taking into account buffer zones and proper nesting.
NOTE: On an overlapping grid you should normally set the errorMask to zero at unused points.

refinementLevel (input) : highest level to refine

baseLevel (input) : keep this level and below fixed, by default baseLevel=refinementLevel-1.

9.29 regridAligned

int

```
regridAligned( GridCollection & gc,
               GridCollection & gcNew,
               bool useErrorFunction,
               realGridCollectionFunction *pError,
               real errorThreshold,
               intGridCollectionFunction & tagCollection,
               int refinementLevel = 1,
               int baseLevel = -1)
```

Access: protected.

Description: Build refinement grids to cover the "tagged points" where the error is greater than an errorThreshold.

Use a bisection approach to build the grids.

refinementLevel (input) : highest refinement level to build, build refinement levels from baseLevel+1,...,refinementLevel

baseLevel (input) : this level and below stays fixed, by default baseLevel=refinementLevel-1 so that only one level is rebuilt.

9.30 splitBoxRotated

int

splitBoxRotated(RotatedBox & box, ListOfRotatedBox & boxList,
realArray & xa, int refinementLevel)

Description: Build possibly rotated boxes

9.31 merge

int

merge(ListOfRotatedBox & boxList)

Description: Attempt to merge rotated boxes.

9.32 regridRotated

int

regridRotated(GridCollection & gc,
GridCollection & gcNew,
bool useErrorFunction,
realGridCollectionFunction *pError,
real errorThreshold,
intGridCollectionFunction & tagCollection,
int refinementLevel = 1,
int baseLevel = -1)

Description: Build refinement grids to cover the "tagged points" where the error is greater than an errorThreshold.

Use a bisection approach allowing rotated grids.

refinementLevel (input) : highest refinement level to build, build refinement levels from baseLevel+1,...,refinementLevel

baseLevel (input) : this level and below stays fixed, by default baseLevel=refinementLevel-1 so that only one level is rebuilt.

9.33 displayParameters

int
displayParameters(FILE *file = stdout) const

Description: Display parameters.

file (input) : display to this file.

9.34 get

int
get(const GenericDataBase & dir, const aString & name)

Description: Get from a data base file.

9.35 put

int
put(GenericDataBase & dir, const aString & name) const

Description: Put to a data base file.

9.36 update

int
update(GenericGraphicsInterface & gi)

Description: Change parameters interactively.

gi (input) :

par (input) :

number of refinement levels :

grid efficiency : a number between 0 and 1, normally about .7

number of buffer zones :

number of refinement levels :

10 ErrorEstimator Reference Manual

10.1 Constructor

ErrorEstimator(InterpolateRefinements & interpolateRefinements_)

Description: Use this class to perform various interpolation operations on adaptively refined grids.

10.2 setDefaultNumberOfSmooths

int

setDefaultNumberOfSmooths(int numberOfSmooths)

Description: Set the default number of smoothing steps for smoothing the error.

10.3 setScaleFactor

int

setScaleFactor(RealArray & scaleFactor_)

Description: Assign scale factors to scale each component of the solution when the error is computed. If no scale factors are specified then the a scale factor will be determined automatically.

10.4 setTopHatParameters

int

**setTopHatParameters(real topHatCentre_[3],
 real topHatVelocity_[3],
 real topHatRadius_,
 real topHatRadiusX_ =0.,
 real topHatRadiusY_ =0.,
 real topHatRadiusZ_ =0.)**

Description: Define the parameters for the top-hat function.

10.5 setWeights

int

setWeights(real weightFirstDifference_, real weightSecondDifference_)

Description: Assign the weights in the error function.

10.6 computeErrorFunction

int

computeErrorFunction(realGridCollectionFunction & error, ErrorFunctionEnum type)

Description: Compute a pre-defined error function of a particular form. These are used to test the AMR grid generator.

twoSolidCircles : error is 1 inside two circles.

diagonal : error is 1 along a diagonal

cross : error is 1 along two diagonals

plus : error is 1 along a horizontal and vertical line, forming a "plus"

hollowCircle : error is 1 near the boundary of a circle.

10.7 computeFunction

int

computeFunction(realGridCollectionFunction & u, FunctionEnum type, real t = 0.)

Description: Evaluate a function that can be used to generate nice AMR grids.

topHat : define a top-hat function

$$u =$$

10.8

int

smooth(realGridCollectionFunction & error)

Description: Apply one smoothing step to the error.

10.9 interpolateAndApplyBoundaryConditions

int

interpolateAndApplyBoundaryConditions(realCompositeGridFunction & error, CompositeGridOperators & op)

Access: protected.

Description: Apply boundary conditions to the error. This will diffuse the error across interpolation boundaries. We do NOT transfer the error from fine patches to underlying coarse patches.

10.10 computeErrorFunction

int

computeErrorFunction(realCompositeGridFunction & u, realCompositeGridFunction & error)

Description: Given a solution u defined at all discretization and interpolation points, define an error function that can be given to the adaptive mesh Regrid function.

u (input) : compute the error from this function.

error (output) : an error function that can be used to perform an AMR regrid.

Notes:

10.11 computeErrorFunction

int

computeErrorFunction(realGridCollectionFunction & u,
realGridCollectionFunction & error)

Description: Estimate errors based on un-divided differences.

$$\frac{c_2}{s_m} \|\Delta_+ \Delta_- u_{i,j}\| + \frac{c_1}{s_m} \|\Delta_0 u_{i,j}\| \quad (1)$$

10.12 computeErrorFunction

int

computeAndSmoothErrorFunction(realCompositeGridFunction & u,
realCompositeGridFunction & error,
int numberOfSmooths = defaultNumberOfSmooths)

Description: Given a solution u defined at all discretization and interpolation points, define an error function that can be given to the adaptive mesh Regrid function.

u (input) : compute the error from this function.

error (output) : an error function that can be used to perform an AMR regrid.

numberOfSmooths (input) : number of times to smooth the error. By default use the default number of smoothing steps (usually 1) which can be set with the `setDefaultNumberOfSmooths` member function.

Notes:

10.13 smoothErrorFunction

int

smoothErrorFunction(realCompositeGridFunction & error,
int numberOfSmooths = defaultNumberOfSmooths,
CompositeGridOperators *op = NULL)

Description: Smooth an error function and interpolate across overlapping grid boundaries.

error (input) : an error function that can be used to perform an AMR regrid.

numberOfSmooths (input) : number of times to smooth the error. By default use the default number of smoothing steps (usually 1) which can be set with the `setDefaultNumberOfSmooths` member function.

op (input) : optionally supply operators to use.

Notes:

10.14 plotErrorPoints

int

plotErrorPoints(realGridCollectionFunction & error,
real errorThreshold,
PlotStuff & ps, PlotStuffParameters & psp)

Description: Plot those points where the error is greater than a threshold.

error (input):

errorThreshold (input) :

10.15 get

int

get(const GenericDataBase & dir, const aString & name)

Description: Get from a data base file.

10.16 put

int

put(GenericDataBase & dir, const aString & name) const

Description: Put to a data base file.

10.17 update

int

update(GenericGraphicsInterface & gi)

Description: Change error estimator parameters interactively.

gi (input) : use this graphics interface.

weight for first difference :

weight for second difference :

set scale factors : Scale each component of the solution by this factor.

11 InterpolateRefinements Reference Manual

11.1 Constructor

InterpolateRefinements(int numberOfDimensions_)

Description: Use this class to perform various interpolation operations on adaptively refined grids.

11.2 setOrderOfInterpolation

int

setOrderOfInterpolation(int order)

Description: Set the order of interpolation. The order is equal to the width of the interpolation stencil. For example, order=2 will use linear interpolation, is second order, and is exact for linear polynomials.

order (input) : the order of interpolation.

11.3 setNumberOfGhostLines

int

setNumberOfGhostLines(int number)

Description: Set the number of ghost lines that are used.

number (input) : the number of ghost lines

11.4 intersects

Box

intersects(const Box & box1, const Box & box2)

Description: Protected routine for intersecting two boxes.

box1, box2 (input) : intersect these boxes.

return value: box defining the region of intersection.

11.5 getIndex

int

getIndex(const BOX & box, Index Iv[3])

Description: Convert a box to an array of Index's.

box (input):

Iv (output):

11.6 getIndex

int

getIndex(const BOX & box, int side , int axis, Index Iv[3])

Description: Convert a box to an array of Index's. Use this version when the box was created with the intersection routine – we need to remove some of the intersection points

box (input):

Iv (output):

side (input) :

//return 0 if the Index's define a positive number of points, return 1 otherwise: //return 0 if the Index's define a positive number of points, return 1 otherwise

11.7

int

interpolateRefinements(const realGridCollectionFunction & uOld,
 realGridCollectionFunction & u,
 int baseLevel = 1)

Description: Interpolate values from the solution on one refined grid to the solution on a second refined grid.

uOld (input): source values

u (output) : target

baseLevel (input) : interpolate values for levels greater than or equal to baseLevel.

11.8 buildBox

BOX

buildBox(Index Iv[3])

Description: Build a box from 3 Index objects.

11.9 buildBaseBox

Box

buildBaseBox(MappedGrid & mg)

Access: protected.

Description: Build a box from a MappedGrid on level=0.

We expand the box on the base level to include ghost points on interpolation boundaries, since we need to allow refinement patches to extend into the interpolation region.

Description: Interpolate the ghost values on refinement grids.

Note: This function assumes that grids are properly nested.

C0 (input) : optionally specify which components to interpolate.

11.15 interpolateCoarseFromFine

int

**interpolateCoarseFromFine(ListOfParentChildSiblingInfo & listOfPCSInfo,
realGridCollectionFunction & u,
int levelToInterpolate = allLevels,
const Range & C0 = nullRange)**

Description: Interpolate coarse grid points that are covered by fine grid points.

C0 (input) : optionally specify which components to interpolate.

12 Interpolate Reference Manual

12.1 Interpolate default constructor

Interpolate()

Purpose: default constructor for the Interpolate class; initialize the class and set default values

Author: DLB

12.2 Interpolate constructor

```
Interpolate(const InterpolateParameters& interpParams_,
            const bool timing_ = LogicalFalse
            )
```

Purpose: constructor for the Interpolate class; initialize the class and set parameter values.

arguments: see the `initialize` member function for a description of the arguments to the constructor

Author: DLB

12.3 initialize

int

```
initialize( const InterpolateParameters& interpParams_,
            const bool timing_ = LogicalFalse
            )
```

initialize the class and set parameter values.

interpParams_: the interpolation parameters are set using the values stored in this object

timing_: if this is set to `LogicalTrue`, timings will be printed out for the initialization step and for each interpolation function.

The following parameters must be set in the `InterpolateParameters` object `interpParams_`:

numberOfDimensions: number of space dimensions

interpolateOrder: the order of interpolation that will be used

interpolateType: see `InterpolateParameters` for choices

amrRefinementRatio(3): can be set through `interpParams_`, but can also be passed into the interpolate functions; the `refinementRatio` can be different in each direction; it must be a power of 2 (including 2^{**0})

Author: DLB

12.4 interpolateFineToCoarse

int

```
interpolateFineToCoarse (realArray& coarseGridArray,
                        const Index Iv[3],
                        const realArray& fineGridArray,
                        const IntegerArray& amrRefinementRatio_ = nullArray
                        )
```

Purpose: interpolate from a fine grid function to a coarse grid function It is assumed that the origin of the `fineGridArray` and `coarseGridArray` are at (0,0,0). Since this is used to compute the location of the interpolatee points, the routine will not compute correct values if this is not the case. Note that for `vertexCentered` grids, polynomial interpolation is pure injection.

coarseGridArray: array to interpolate to (“interpolation” points); stride 1 required note that a `realMappedGridFunction` can be passed in here since it is a derived class from `realArray`

Iv[3]: defines the target interpolation points. They are given by `fineGridArray(Iv[0],Iv[1],Iv[2])`

fineGridArray: array to interpolate from (“interpolee” points); stride 1 required note that a `realMappedGridFunction` can be passed in here since it is a derived class from `realArray`

amrRefinementRatio(3): IntegerArray containing refinementRatio in each of the three dimensions; if `nullArray` is passed in, `amrRefinementRatio` defaults to the values set upon instantiation of the class

Author: DLB

12.5 interpolateCoarseToFine

int

```
interpolateCoarseToFine (realArray& fineGridArray,
                        const Index Iv[3],
                        const realArray& coarseGridArray,
                        const IntegerArray& amrRefinementRatio_ = nullArray,
                        const InterpolateParameters::InterpolateOffsetDirection*
                        interpOffsetDirection =NULL
                        )
```

Purpose: interpolate from a coarse grid function to a fine grid function It is assumed that the origin of the `fineGridArray` and `coarseGridArray` are at (0,0,0). Since this is used to compute the location of the interpolatee points, the routine will not compute correct values if this is not the case.

Algorithm: This routine uses the standard Lagrange interpolant formula. Since the interpolation occurs at regularly-spaced points, the computation of the interpolation coefficients can be done mostly with integer arithmetic. The routine is optimized so that the general interpolation formula is not used when, e.g. the interpolation points lie on coarse grid lines or at coarse grid points. Instead, only the non-zero coefficients are explicitly used in the computations.

fineGridArray: array to interpolate to (“interpolation” points); stride 1 required

Iv[3]: defines the target interpolation points. They are given by `coarseGridArray(Iv[0],Iv[1],Iv[2])`

coarseGridArray: array to interpolate from (“interpolee” points); stride 1 required

amrRefinementRatio(3): IntegerArray containing refinementRatio in each of the three dimensions; if nullArray is passed in, amrRefinementRatio defaults to the values set upon instantiation of the class. amrRefinementRatio(i) must be a power of 2

interpOffsetDirection[3]: for odd-order interpolation, the stencil must be offset to one side or the other. interpOffsetDirection[axis] specifies in which direction interpolation in the axis direction will be offset. If interpOffsetDirection is NULL, the default value InterpolateParameters::defaultInterpolateOffsetDirection will be used in all directions.

000630: Currently the default value of InterpolateParameters::defaultInterpolateOffsetDirection is offsetInterpolateToLeft

Author: DLB

13 InterpolateParameters Reference Manual

13.1 InterpolateParameters default constructor

InterpolateParameters(const int numberOfDimensions_, const bool debug_)

Purpose: default constructor for the InterpolateParameters container class; initialize the class and set default values

13.2 InterpolateParameters destructor

InterpolateParameters()

Purpose: destructor for the InterpolateParameters container class

13.3 setAmrRefinementRatio

void

setAmrRefinementRatio (const IntegerArray& amrRefinementRatio_)

Purpose: set InterpolateParameters::amrRefinementRatio

amrRefinementRatio_: the value in amrRefinementRatio_(axis) is the refinement ratio in the “axis” direction. It is a positive number equal to the number of fine grid points per coarse grid point in this direction; the Interpolate class functions are only implemented for values of amrRefinementRatio that are a power of 2

13.4 setInterpolateType

void

setInterpolateType (const InterpolateType interpolateType_)

Purpose: set InterpolateParameters::interpolateType

interpolateType_: is used to set the type of interpolation. It can be chosen from enum `InterpolateType` defaultValue, polynomial, fullWeighting, nearestNeighbor, injection, numberOfInterpolateTypes ;

000623: currently only polynomial interpolation is implemented in the `Interpolate` class

13.5 numberOfDimensions

```
void
setNumberOfDimensions (const int numberOfDimensions_)
```

Purpose: set `InterpolateParameters::numberOfDimensions`

numberOfDimensions_: since the `Interpolate` functions only deal with `realArray` 's, they have no way of knowing what the dimension of the problem is. This parameter is used to set that value.

13.6 setInterpolateOrder

```
void
setInterpolateOrder (const int interpolateOrder_)
```

Purpose: set `InterpolateParameters::interpolateOrder`

interpolateOrder_: this is the order of interpolation that will be used by the interpolation functions in the `Interpolate` class. It must be set initially because the interpolation stencil size is determined from it, and is used in the precomputation of the interpolation coefficient matrix.

13.7 setGridCentering

```
void
setGridCentering (const GridFunctionType gridCentering_)
```

Purpose: set `InterpolateParameters::gridCentering`

gridCentering_: this parameter is used to tell what kind of centering is used on the underlying grid. Again, since the `Interpolate` class doesn't see anything but the `realArray` 's it can't tell what the centering of the mesh was.

13.8 setUseGeneralInterpolationFormula

```
void
setUseGeneralInterpolationFormula (const bool TrueOrFalse = LogicalFalse
)
```

Purpose: set `InterpolateParameters::UseGeneralInterpolationFormula`. If set to `True`, the interpolation will be computed using the general formula rather than the "optimized" explicitly written-out formula for lower interpolation orders

000626: N.B. The general interpolation formula is actually optimized for the cases where some of the interpolation coefficients are zero, i.e. it doesn't multiply by those coefficients. The explicit formulas are not currently optimized for these special cases

13.9 interactivelySetParameters

int

interactivelySetParameters ()

Purpose: interactively set InterpolateParameters parameters using NameList

13.10 amrRefinementRatio

int

amrRefinementRatio (const int axis) const

Purpose: return component of InterpolateParameters::amrRefinementRatio

axis: refinement ratio for the `axis` direction will be returned

13.11 gridCentering

GridFunctionType

gridCentering () const

Purpose: return the gridCentering

13.12 useGeneralInterpolationFormula

bool

useGeneralInterpolationFormula () const

Purpose: return the value of useGeneralInterpolationFormula

13.13 interpolateType

InterpolateType

interpolateType () const

Purpose: return type of interpolation that will be used

13.14 interpolateOrder

int

interpolateOrder () const

Purpose: return the order of interpolation that will be used

13.15 numberOfDimensions

int

numberOfDimensions () const

Purpose: return value for numberOfDimensions

References

- [1] J. BELL, M. BERGER, J. SALTZMAN, AND M. WELCOME, *Three dimensional adaptive mesh refinement for hyperbolic conservation laws*, SIAM J. Sci. Comput., 15 (1994), pp. 127–138.
- [2] M. BERGER AND I. RIGOUTSOS, *An algorithm for point clustering and grid generation*, IEEE Trans. Systems Man and Cybernet, 21 (1991), pp. 1278–1286.
- [3] M. J. BERGER, *Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations*, PhD thesis, Stanford University, Stanford, CA, 1982.
- [4] M. J. BERGER AND P. COLELLA, *Local adaptive mesh refinement for shock hydrodynamics*, J. Comp. Phys., (1989), pp. 64–84.
- [5] M. J. BERGER AND J. OLIGER, *Adaptive mesh refinement for hyperbolic partial differential equations*, J. Comp. Phys., 53 (1984), pp. 484–512.

Index

AMR

- error estimation, 11

- regridding, 8

- time stepping, 6

- amrHype, 6

error estimator

- parameters, 33

interpolation

- on an AMR grid, 13

show file

- options, 29